AN OBJECT ORIENTED APPROACH

TO INVENTORY MANAGEMENT SYSTEMS

by

Antonio Maria Palma dos Reis

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Management Information Systems

at

The University of Wisconsin-Milwaukee

May 1993

AN OBJECT ORIENTED APPROACH

TO INVENTORY MANAGEMENT SYSTEMS

by

Antonio Maria Palma dos Reis

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Management Information Systems

at

The University of Wisconsin-Milwaukee

May 1993

AN OBJECT ORIENTED APPROACH

TO INVENTORY MANAGEMENT SYSTEMS

by

Antonio Maria Palma dos Reis

The University of Wisconsin-Milwaukee, 1993

Under the Supervision of

Professor Doctor William D. Haseman

With the increasing market competition and rate of change, managing inventories as efficiently as possible is critical for organizations holding products for sale.

The diversified patterns of demand and storage characteristics of the products require the inventory management systems to take into account the specificities of each product or product line.

Inventory management systems were developed in several environments, from the third generation languages handling transaction files to the database environments.

Each approach has some good features, but also some shortcomings. The third generation languages are likely to provide good processing speed and an efficient use of hardware, but lack of flexibility and difficult maintenance are likely to be a problem.

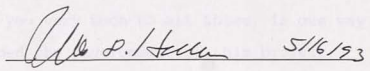
The database approach, requiring more hardware resources, may provide data sharing across applications, in a much more flexible and maintainable environment. However, database applications may run at a very moderate speed, and, in the specific case of inventory management, the product behavior type must be passed from the data to the application, producing some lack of data independence.

A model was developed to build an inventory management system using the inheritance and incremental specialization concepts available in the object oriented paradigm. This approach should allow increased flexibility and expendability, since a new product may simultaneously inherit some data and behavior from an existing product and specialize some other data elements or behaviors.

The actual implementation of the prototype was developed in C++. It worked properly, executing the inheritance and incremental specialization as expected. However, the initial work to establish the base classes was

significant, leaving the suggestion that this approach is more applicable to mid size to large systems.

Overall, the object oriented approach seemed to be a promising solution for this kind of problem, specially for large systems. The current complexity of data management and instance identification might be solved by the developing object oriented databases.



Major Professor Date

ACKNOWLEDGMENTS

I would like to thank all the support from my academic adviser in the United States, Professor Doctor William D. Haseman, from my adviser in my home country, Portugal, Professor Doctor Amilcar S. Gonçalves, and from Professor Doctor Raims, as well as from all other faculty members that helped the development of this project.

I would also like to thank the financial support of INVOTAN (NATO), of the Instituto Superior de Economia e Gestão and of the Portuguese Fulbright Commission.

Thank you very much to all those, in one way or the other, helped the development of this project.

TABLE OF CONTENTS

LIST OF FIGURES

I - INTRODUCTION - Inventory Decisions Facing Different Item Behaviors

In todays competitive environment, where the rate of change keeps increasing, managing inventories properly is critical for organizations holding products for sale.

The behavior of inventory items depends on the product features, the market environment, and the consumer attitudes and perception towards the product.

Some product features influencing the inventory behavior are the life of the product and the technological obsolescence. Bread and vegetables, for example, have a short life - just some days - while house supplies may be sold over several years. In some products, like computers, VCRs, or televisions, the technological obsolescence may reduce the product value more than the physical deterioration.

Other important product feature is the storage cost. Smaller items are likely to be less expensive to store, and items requiring special care are likely to be more expensive to store. For example, storage cost of pets is likely to be a significant part of the cost structure of a pet shop.

The market environment, as described by Michael Porter competitive analysis, may change the firm's competitive position related to a specific product or class of products, therefore affecting consumer behavior. For example, when a competitor begins offering a similar product to the same market or in the same region, the demand, even if formerly was behaving in a deferred sales basis[1], is likely to begin behaving in a lost sales basis[2].

One other possible disturbance in the market environment may be the possible lack of suppliers or delay in supplies. In some circumstances, the supply of some items may be strongly reduced, or the waiting time for them increased. Very bad agricultural years may reduce the availability of produce or, when a car manufacturer produces a new very innovative model, dealers may have to wait a fair amount of time to get their orders satisfied.

Even in regular circumstances, some items have a systematic delay in delivery. That happens when the supplier instead of stocking the item produces or orders it in a special order basis.

---

[1] By deferred sales basis is meant that, if the customer searches for the item and it is out of stock, he is likely to wait for the stock to be renewed and, after that, buy the item at the same store.

[2] By lost sales basis it is meant that, if the customer tries to buy an item out of stock, he would not wait and come back to buy it later.

The consumer attitudes and perceptions about a product may define it as unique, and so accepting to wait for it if it is out of stock - deferred sales - or as having substitutes. In the later case, the customer would not be likely to be willing to wait for the new stock. The consumer may also be more willing to buy that item in some occasions, producing seasonality. Toys are likely to sell much better in the pre-Christmas season than in other times of the year, and the sales of beer are likely to go up in the summer.

Taking into account all these variables, inventory managers have to decide, within very short time frames, for each product, when to place the order, and for how much quantity. The inventory manager also has to execute some product and supplier selection, but those decisions are outside the scope of this work.

The major goal of this study would be to ascertain to what extend and in what ways may the Object Oriented analysis and programming improve the inventory management performances in taking the former decisions.

II - The Former IS Approaches

The traditional approach to inventory management would be to use third generation languages handling files. These files were not likely to be normalized. The typical solution would be to have a static file of products and a transaction file that would be runt against the master (or static) file to determine the current position and, eventually, compute the economic order quantity for an order in that product.

This approach had a lack of flexibility and data independence. The lack of data normalization[3] would produce add, update and deletion anomalies. The different fields of the files being likely to be defined by character counting and the lack of data normalization, would made data and programs tightly dependant on each other. That means that it would be very difficult to use the existing data in new

---

[3] By data normalization is meant that the files should respect the third normal form rules.

In order to be in the first normal form, there should be no repetitive groups on the file. For the second normal form it is required to be in the first normal form and to have no functional dependencies. Functional dependencies are fields (attributes) that depend on only part of the composite key. In order to be in the third normal form, a file must be in the second normal form and have no transitive dependencies, i.e., no attributes depending on other attributes that are not part of the key.

I am not including the Boyce-Codd normal form and the Fifth normal form in this concept of data normalization.

applications, and that changes in the application could very easily require changes in the data files.

With the urge in the current business environment for data sharing across the company, and the need for flexible solutions in order to decide and act as speedy and accurate as possible, this kind of solution does not look like a very appropriate one.

The traditional approach has, however, some good features. Comparing to relational databases or to object oriented programming, third generation languages handling files are likely to require a smaller amount of hardware resources, such like CPU speed and capacity. One other advantage, in the perspective of some companies, is the fact that these systems are already installed, so there is no implementation (investment) cost required to operate them. On the other hand, maintenance cost may be significant.

More recently, relational databases had been used to solve managerial problems using wide sets of data. This approach offers excellent sharing of data, since new applications may use the existing data very easily.

The different items behavior is likely to be implemented by inclusion of flag attributes to define behavior. The applications, apart from the data, are likely

to select the appropriate economic order quantity functions and procedures. However, the transfer of control information, in the form of a flag, from the data to the applications, produces some lack of independence between data and programs.

III - The Object-Oriented approach

The Object-Oriented (O-O) approach claims to provide a mechanism to better formalize a model of reality. Object-Oriented methodologies allow for the construction of complex, inherently reusable systems. The O-O systems also have lower life-cycle costs due to increased programmer productivity and reduced maintenance. (LeClaire 1992)

The O-O approach presents as distinguishing features the polymorphism, the encapsulation, the inheritance, and the dynamic binding.

Polymorphism allows different (and dissimilar) objects to be treated in a similar way, i.e., to be sent the same message, once they have compatible protocols.

Objects are identifiable entities that may be composed of data and behavior. So objects, as opposed to data, are not inert matter.

The objects' behavior is defined by its own methods. Once the object is sent a message, if the sender has the permission to order that message's method, the method would be executed, acting the respective behavior.

Object communication is implemented by sending messages to one another as a way of completing their tasks.

Encapsulation is the objects ability to construct a protective barrier around themselves, selecting which other objects or classes of objects are allowed to call which methods.

Inheritance is the ability to receive data and behavioral information from the more general classes. Once a class is a subclass of some other superclass, the attributes and the methods of the superclass are inherited by the subclass. This builds an inheritance hierarchy.

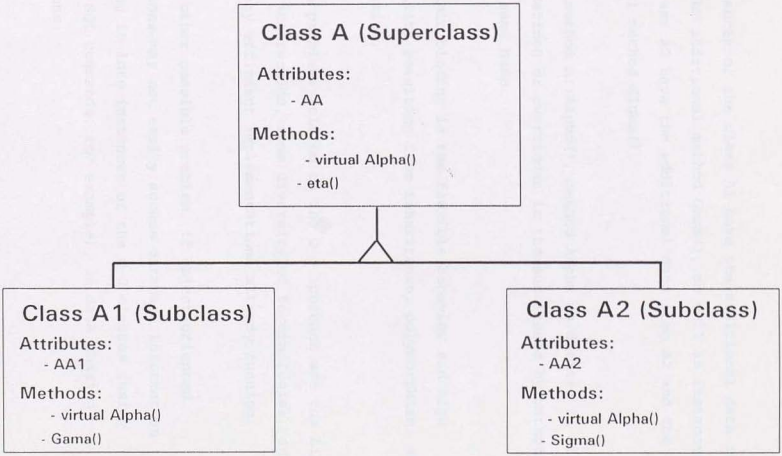
The inheritance of the more general data items and behaviors, and the inclusion of the more detailed items in the subclasses builds an incremental specialization through the hierarchy tree.

Inheritance produces incremental specialization, down the hierarchy tree, from superclasses to subclasses. To document inheritance and incremental specialization, the graph number III.1 was created.

Instances of the classes A1 or A2 (subclasses) include the data item AA and the methods Alpha() and Beta() inherited from the superclass A.

III.1 -Inheritance and Incremental Specialization

(Conceptual example)



Instances of the class A1 have the additional data item AA1 and the additional method Gama(), as well as instances of the class A2 have the additional data item A2 and the additional method Sigma().

The method A::Alpha(), method Alpha as defined in class A, is redefined or overridden in the subclasses by methods with the same name.

Dynamic binding is the flexible behavior and high expendability resulting from inheritance, polymorphism, and encapsulation.

Some problems pointed to the O-O approach are the high level of abstraction, some diversity of terminologies, and not to many efficient implementations already running.

One other possible problem, if object-oriented applications may not easily access database information retrieving it into instances of the O-O classes (using embedded SQL commands, for example), is data sharing limitations.

IV - A generic Object-Oriented (O-O) Model

Developing a generic O-O model, correspondences between classes and groups of items should be found.

The classes considered were the following:

- Generic product: Defines the data and behavior of the products following the Economic Order Quantity (EOQ) as defined by the Wilson model;

- Quantity discounts: This class stores the percentage of discount and the minimum order required to access the discount for a certain product. It interacts with the virtual function[4] EOQ() to find the optimum quantity to be ordered;

- Certain demand: Virtual class[5] to classify the products as opposed to uncertain demand products;

- Uncertain demand: Class to store the attributes defining the distribution of demand, such as expected value

---

[4] By virtual function it is meant that, if the function exists in the subclass of the acting object, the function in the subclass overrides the function in the superclass, so the subclass behavior is implemented, even if the pointer calling the function points at the superclass component of a subclass instance.

By object is meant a particular instance of a class.

[5] A virtual class is a class that does not store data or implement new or specialized methods. The purpose of this type of classes is to organize the class diagram, and very often, they may be omitted, improving performance.

and standard deviation. This class could have subclasses implementing EOQ models for each specific statistical distribution. The subclasses would store as data items the parameters not included in the superclasses;

- Stable demand: Virtual classes to classify the products as opposed to seasonal demand products;

- Seasonal demand: Class to specify seasonality factors, such as seasonality type, which may be additive or multiplicative, seasonality period, i.e., 4 quarters, 12 months, 54 weeks, 365 days, etc., and the seasonality factors for each period;

- Backlogged demand: Class to implement the ability to backlog orders.

The refined generic O-O model was derived removing the virtual classes from the O-O model.

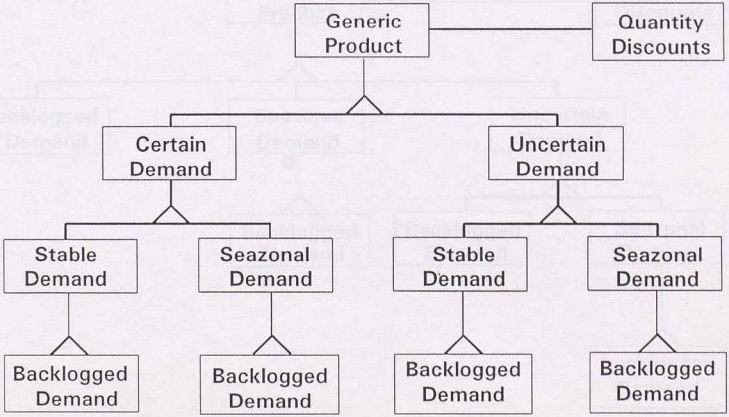
Incremental specialization, as defined in chapter III, is clear in this model since the subclasses detail the data and behavior of the superclasses.

Data is widely inherited through the class hierarchy. All subclasses inherit the data attributes of the generic product, as well as some of the functions.

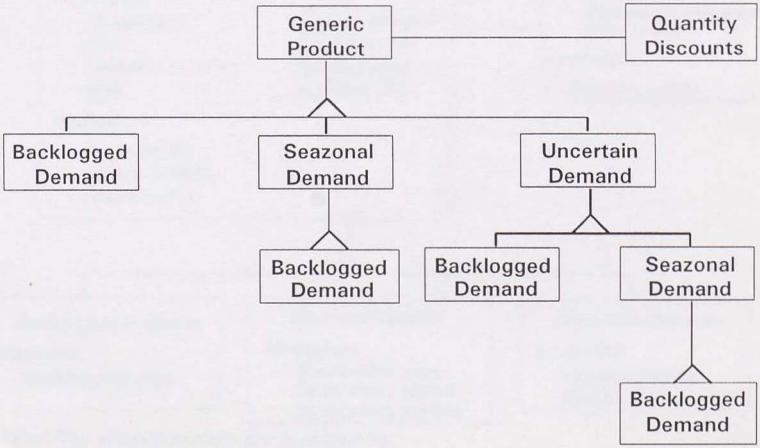
Most of the functions are virtual functions, so they are likely to be overridden by functions with the same name (protocol) in the subclasses.

V.1 - A generic O-O model

Class hierarchy

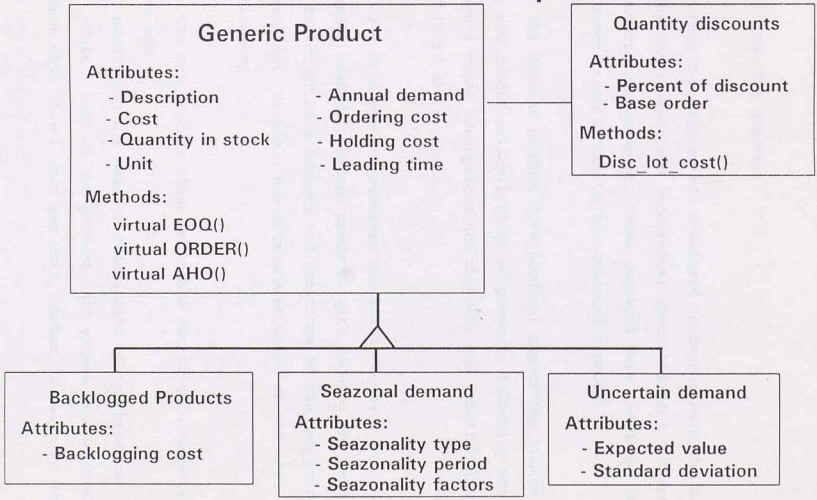


V.2 - Refined generic O-O model



V.3 - A generic O-O model

(Inheritance and Incremental Specialization)



Note: The virtual functions are overriden by similar functions in the subclasses

V - O-O Model for a Subset of the Situations

. The O-O diagram

This O-O diagram was developed to demonstrate the use of the inheritance and incremental specialization concepts in inventory management. These concepts were discussed in the description of the Object Oriented approach.

The Generic product (Gen_product) superclass stores the data and functionality typical of products following the Wilson's model assumptions and the data and functionality generic to all products.

The Back_product subclass specifies the data and behavior needed to manage products with backlog and inherits all the other data members and functions of the Gen_product superclass. It has a new data member which is the cost of backlogging.

The Gen_product class defines a set of data members, which are:

pDescription - Name or description of the product;

pUnit - Unit of the product. All values are defined per product unit, i.e., cost per unit, number of units to order, etc;

pNumber - Product number or identifier;

pCost - Acquisition cost of one unit of the product;

pStock - Quantity in stock at the current time. Assuming no backlogged orders this value may not be negative;

pDemand - Quantity demanded per time period[6], one year, for example;

pOrder_cost - Cost of issuing an order for the product;

pHold_cost - Cost of holding one unit of the product during one period;

pSavety - Minimum level of stock during any cycle. The new order should arrive when the stock reaches this level;

pinOrder - Quantity of the product ordered but not received yet;

pLead - Fraction of the period happening between the issuance of the order and the reception of the merchandise.

---

[6] In order to have the functions operating properly, a period, as a stable amount of time, must be defined. Usually, the period is likely to be one year, but it may be a month, a semester, etc. The important point is that the period considered must be consistent in all values.

For example, if the period taken into account for the estimation of demand is one year, the fraction of leading time must be computed as leading time as a fraction of one year.

. **The functionality - Gen_product superclass**

A set of constructors, destructors, overloaded operators and retrieve and update functions for the data members are defined in the programs, but due to the technical character of these functions, they are discussed in the C++ implementation section.

A set of virtual functions is defined to provide the functionality needed to execute the inventory management. The use of virtual functions provide easier incremental specialization, as discussed in the C++ implementation section.

These are:

- sale(int quantity) - Receives as a parameter the amount sold and deducts it from the stock. If the quantity sold is greater than the existing stock, an error message is displayed and the sale is canceled. Since this function is virtual, it may be easily overridden by new functions in the subclasses to allow for new actions or different behavior in the subclasses. One example is in the Back_product subclass, where the sale(int quantity) function allows for sales beyond the current level of stock (backlogged sales).

This function also calls the order() function which testes if a new order should be issued due to the lower level of stock;

- supply(int quantity) - Receives the quantity delivered as parameter and updates the quantity in stock and the pending orders. If the pending orders are not enough to support the whole supply, a zero is stored in the pending orders data member;

- display(void[7]) const[8] - Displays all the relevant information of the product. All data members are displayed as well as the results of some member functions, as the economic order quantity, and the annual holding costs. This function is virtual in order to be easily overridden by similar functions in the subclasses displaying all details of the specific products (Incremental specialization);

---

[7] A function receiving "void" as parameter is a function that needs no parameter at all.

Void is a way of stating that that function receives no parameter and no mistake was done on forgetting to type in the parameters.

[8] A "const" function is a function that does not change the object in any way. For example, the display() function displays the object but does not change the status of any data member, so it is a "const" function.

"Const" may also be used as a parameter qualifier, but in that situation it means that the parameter is not changed when used in the function.

- eoq(void) const - Computes the Economic Order Quantity (EOQ) according to the Wilson Model and returns its value;

- order(void) - Checks if an order should be issued at the current level of stock. In order to do that, this function takes into account the current stock, the quantity in pending order, the safety level of stock, the leading time, and the estimated quantity demanded by period. If a new order should be issued, this function calls the eoq() function to compute the amount to order and adds the quantity ordered to the pending orders. If the system would be connected to an automatic ordering system, this function would call the order issuance and sending[9] functionality. This function is also a virtual function to allow for easier incremental specialization;

- aho(void) const - This virtual function computes and returns the annual inventory costs of this product. These costs, in the Wilson Model, include the ordering costs, and the costs of holding the merchandise. In more complex models, some other costs may need to be taken into account, but the function may be overridden by specialized functions.

---

[9] The technology recommended for order issuance and sending would be the use of a faxboard. With that technology, no human effort would be required to send an order, the system would know the faxnumber of the supplier and fax the order to him.

. **The functionality - Back_product subclass**

The Back_product subclass inherits most of the data and functionality from the Gen_product superclass.

Data members like pNumber, pCost, pStock, pDemand, pOrder_cost, pHold_cost, pinOrder, and pLead are simply inherited from the superclass.

The retrieve and update functions for the data members of the superclass and the superclass function supply(int quantity) are also inherited from the superclass.

Some other functions, existing in the superclass, are overridden to specialize the behavior of the backlogged products.

These are:

- sale (int quantity) - In the backlogged products subclass, since backlogging is allowed, it is possible to sell beyond the stock level, so the sale(int quantity) method does not check for negative stocks;

- display(void) - The specialized display function displays also the new data member back_cost[10] and the results of the function dif(void) which returns the quantity to be backlogged in each cycle;

- eoq(void) - Adjusts the Wilson's Economic Order Quantity to the possibility of backlogging orders, taking into account the cost of backlogging;

- order(void) - Checks whether or not an order should be placed at the current level of stock. This function takes into account the stock level, the quantity in pending orders, the optimal quantity to be backlogged (method dif(void)), the leading time and the demand per period. If an order should be placed, it calls the eoq(void) function and adds the Economic Order Quantity to the pending orders. Just like the equivalent function in the superclass, if there would be an automatic ordering system, this function should execute the order;

- aho(void) - Computes the global inventory costs taking also into account the backlogged products and respective backlogging costs.

---

[10] Cost of backlogging one unit of the product for the next cycle, i.e., cost of supplying the item to the customer not with stock of the current inventory (since we are out of stock) but with stock to be received in the next delivery from our supplier.

Two new virtual functions are added to the generic product. These are the optimal amount of deferred sales per cycle (dif(void)), and the maximum level of stock (maxStk(void))

Some new member functions are required for the operation of the subclass. These are the constructors, destructors and overloaded equal operator of the Back_product class, as well as the cost of backlogging retrieve and update functions. These functions will be discussed in the next section due to its technical character.

VI - Implementation in C++

This section discusses physical issues of the current implementation.

These are the file organization, constructors, destructors, overloaded equal operators, the product link list, the data members update and retrieve functions, and the implications of the use of virtual functions.

. File organization

This program is organized as a C++ project. The project compiles and links three main files.

These are:

- InvModel.cpp - Prompts the menu to the user and calls the methods defined in the classes. If adding new subclasses this file may need to be changed to prompt the user the appropriate questions and options;

- Gen_Prod.cpp - Defines the methods inherent to generic products. This methods may be inherited by specialized kinds of products, without any change in this file.

- Strings.cpp - Defines the methods operating on the string class. By including this file, string input, output and several other operations on strings may be done by calling a function or using an overloaded operator, what gets string management easier;

- Bac_Prod.cpp - This file, specializing the behavior of the Gen_product class, was included to document the abilities of inheritance and incremental specialization. This file contains the methods operating on the Back_product class that differ from the methods used by the Gen_product class. The methods that do not differ from the Gen_product class (superclass) are inherited from the superclass.

The project also includes some header files[11].

These are:

- Gen_Prod.h - Header file with the String and Gen_product classes declarations, as well as the Boolean and Relational enumerations.

- Bac_Prod.h - Header file corresponding to the Bac_Prod.cpp file. In this header file the Gen_product class and its data members and methods are declared.

---

[11] Header files are files that declare the classes, its data members and methods. By reading the header file, it is possible to roughly understand the behavior of the class.

. Constructors

Each time an instance of a class is created, the constructor for that class must be activated. The Gen_product and the Back_product classes have two kinds of constructors.

The first constructor, builds the instance from scratch, i.e., receives a set of character pointers and numbers and builds a new instance of the class. Since default values are defined in the declarations in the header files, if the number of parameters used when calling the constructor is not enough, the default values are used. Taking the lack of parameters in the function call to an extreme, if the constructor would be called without any parameter, an empty instance would be created with all numeric data members set to zero.

The second constructor is a copy constructor. It receives as parameter the reference of another instance of the same type and replicates it. This constructor is not being used in the InvModel.cpp code, but it may be useful to support further functionality.

When a constructor of a subclass is called, it calls the constructor of the superclass, creating a composed instance, i.e., an instance that has a subclass component and a superclass component. The existence of these two components creates the possibility of having two different addresses, and, consequently the possibility of pointers with different values, to the same instance of the class. One would be the pointer to the superclass component and the other the pointer to the subclass component.

The Gen_product constructors add the product instance to a two-way link list. This link list is not used by the current inventory management module (InvModel) but it may be useful for possible extensions.

. Destructors

As well as the constructors, the execution of a subclass destructor activates the superclass destructor to destroy the superclass component of the object. So, when the Back_product destructor is activated, it destroys the subclass component of the object and calls the Gen_product destructor to destroy the superclass component of the object and update the Gen_product link list.

. Overloaded equal operators

C++ allows to overload operators for user defined classes. The equal (=) operator does not know how to behave towards instances of Strings, Gen_products, or Back_Products, so, if we want to assign the values of an instance of one of these classes to another instance of the same class, we must redefine, i.e. overload, the equal operator.

So, the overloaded equal operators are a redefinition of the equal operator to handle instances of these classes.

Once the equal operator is overloaded, the behavior produced by the operator (=) depends on the class of the instances being assigned. If the instances are integers or floats, a simple copy of the number is executed, if the instances are instances of the user defined classes with overloaded equal operators, the behavior (data members to be assigned) is defined by the user.

. The product link list

A two-way link list is maintained by the constructors and destructors. The static components[12] of the link list are a pointer to the first instance and a pointer to the last instance.

The non-static components, components depending on each particular instance, are the pointer to the previous instance and the pointer to the next instance. In the extremes these pointers may have NULL values.

. The data members update and retrieve functions

Since the data members are private attributes of the object, they may not be accessed by other object's methods. To give access to other object, a public function must be defined, and the kind of access granted is the actions allowed by the function.

---

[12] Static members of a class are members that are unique for the whole class, they do not depend on the different class instances.

Static data members, as the pointer to the first instance and the pointer to the last instance exist at the level of the class, each instance does not hold or own them.

Static member functions are functions that operate with no specific instance, so they do not require the reference or pointer to a specific instance to be activated.

This procedure is an example of dynamic binding. The private members, which are not accessible by default, may be accessed (retrieved or updated) by object member functions created with the purpose of allowing these transactions.

The retrieve and update functions, for a specific data member, have the same name, they just differ in the "const" qualifier (discussed earlier) and in the parameters received.

The retrieve functions are "const",i.e. they do not change the object, and receive no parameter. The update functions may not be "const", since they update the object, and receive the new value of the data member as parameter.

. Implications of the use of virtual functions

As discussed before, inheritance allows subclasses to inherit data members and methods from the superclasses and, the existence of a superclass component and a subclass component of the same object, allows the object to be referenced or pointed to with two different addresses, the address of the superclass component, or the address of the subclass component.

If the current object[13] is an instance of a subclass and there are methods in both the subclass and the superclass with the same name, returning a value or reference of the same class, and using the same parameters, the compiler has to decide which method to execute, the superclass method or the subclass method.

In this situation, if the current pointer or reference points to the subclass component of the object[14], the subclass method is going to be executed.

If the pointer or reference points to the superclass component of the object, the decision depends on whether or not the function is a virtual function. In the case of a virtual function, even if the pointer being used points to the superclass component of a subclass object, the subclass method is executed. If the function is not virtual, the superclass method is executed.

The use of virtual functions enhances the incremental specialization abilities. For example, the set of pointers to Gen_products in the link list, may call the same method

[13] The current object is the object pointed by the pointer or reference used to access the method.

[14] Instances of the subclasses have a subclass component and an inherited superclass component. So, it is possible to point to the object by pointing to one component or to the other.

for each product (eoq(), for example) and the virtual feature of this function allows each product to behave according to its subclass method.

C++ allows the use of pure virtual functions. These functions not only give priority to subclass methods, but also require any defined subclass to redefine the function. In this case, the subclasses are not allowed to simply inherit the method from the superclass.

VII - Conclusions

The development of this prototype gave evidence to some good and some not so good features of C++, and, in some way of the object oriented approach.

The convenience of inheritance and incremental specialization seems very real, especially in large or growing systems. The ability to inherit data and methods and to specialize them suggests a more effective expendability and maintenance.

On the other hand, for small systems, or for systems where expendability does not seem to be required, the initial efforts of declaring classes, constructors and destructors have a doubtful payback. The cost of developing a small system in conventional third generation languages seems to be lower, and the benefits, if expendability is not an issue, are likely to be similar.

One other issue is the data management difficulty. C++ uses pointers and references as the major data management and instance identification tool. The possibility of pointers with different values pointing at different components of the same object further complicates the use of pointers already not easy to learn to junior programmers.

Comparing the difficulty and technical nature of the use of pointers and references with the much more understandable data management techniques of the relational model, in a real inventory management systems development, I would be likely to chose to develop this kind of package in a relational database due to the heavy data use and the importance of data sharing.

To solve the data management difficulties, new object oriented databases are being developed based on the object oriented languages. Examples are Gemstone, based on Smalltalk, VBase, based on C++, and others.

Overall, the C++ implementation of the Object-Oriented approach, seemed to be conceptually very valuable and well structured. At the data management level, it seems to me that some improvement could be done towards getting simpler and more natural ways of storing, retrieving and identifying instances. Actually, the new Object-Oriented databases are evolving towards solving the data management issues.

```
/*
     Program:  INVMODEL.CPP
     Author:        Antonio MP Reis
     Date:          April, 20, 1993
     Purpose:  Run the various inventory models

*/

#include <iostream.h>

#include "c:\trab\thesis\gen_prod.h";
#include "c:\trab\thesis\bac_prod.h";

int menu(void)
     {
     int selection;
     do
          {
          cout << "\n\tOptions:\n"
               << "\t\t(1)\tAdd new product"
               << "\n\t\t(2)\tGet product information"
     << "\n\t\t(3)\tExecute sale\n\t\t(4)\tAccept supply"
               << "\n\t\t(5)\tCheck EOQ"
               << "\n\t\t(6)\tUpdate product data"
               << "\n\n\t\t(7)\tQuit\n\t\t";
```

```
        cin >> selection;
        }
    while (selection < 1 || selection > 7);
    return (selection);
    }


void runMenu(Gen_product *aProduct)
    {
    int selection, quantity;

    do
        {
        selection = menu();
        switch(selection) {
        case 1:
          cout << "\n\nNot done yet - This version may "
             << "handle only one product\n";
          break;
        case 2:
          aProduct->display();
          break;
        case 3:
          cout << "\n\nEnter number of units sold\n";
          cin >> quantity;
          aProduct->sale(quantity);
          aProduct->display();
```

```
            break;
          case 4:
            cout << "\n\nEnter number of units received\n";
            cin >> quantity;
            aProduct->supply(quantity);
            aProduct->display();
            break;
          case 5:
            cout << "\n\n\tThe Economic Order is\t" <<
                    aProduct->eoq() << '\n';
            break;
          case 6:
            cout << "\n\nNot done yet";
          };
          }
     while (selection != 7);
     return;
     }

main()
     {
     char aString[20];
     int aCost, aStock, aDemand, aOrder_cost, aHold_cost,
aSafety;
     int aBack_cost;
     float aLead;
     Gen_product *aProduct;
```

```
    cout << "\n\n\tEnter the product name . . .\t";
    cin >> aString;
    cin.ignore();
    String aDescription(aString);

    cout << "\n\tEnter the product unit . . .\t";
    cin >> aString;
    cin.ignore();
    String aUnit(aString);

    cout << "\n\tEnter the product cost . . .\t";
    cin >> aCost;

  cout << "\n\tEnter the product current stock .\t";

    cin >> aStock;

    cout << "\n\tEnter the cost of backlogging\t"
<< "\n\t(Zero if backlogging is impossible). . .\t";

    cin >> aBack_cost;
```

```
        if (aBack_cost == 0)
           {
      cout << "\n\tEnter the safety level of stock..\t";
           cin >> aSafety;
           }

cout << "\n\tEnter the product demand per period .\t";
        cin >> aDemand;

cout << "\n\tEnter the cost of placing an order .\t";
        cin >> aOrder_cost;

        cout << "\n\tEnter the holding cost "
             << "\n\tper product/period . . . .\t";
        cin >> aHold_cost;

        cout << "\n\tEnter the lead time "
             << "\n\tas a fraction of period . . .\t";
        cin >> aLead;
```

```
        if (aBack_cost == 0)
                aProduct = new Gen_product(aDescription,
aUnit, aCost, aStock, aDemand, aOrder_cost, aHold_cost,
aLead, aSafety);

          else

                aProduct = new Back_product(aDescription,
aUnit, aCost, aStock, aDemand, aOrder_cost, aHold_cost,
aLead, 0, 0, aBack_cost);
    aProduct->display();
    runMenu(aProduct);
    }
```

```
/*
     Program:  GEN_PROD.CPP
     Author:        Antonio MP Reis
     Date:          April, 20, 1993
     Purpose:  Supporting the general product behavior

*/

#include "c:\trab\thesis\gen_prod.h"


Gen_product *Gen_product::firstProd;
Gen_product *Gen_product::lastProd;


Gen_product::Gen_product(const char *aDescription, const
char *aUnit, const int aCost, const int aStock, const int
aDemand, const int aOrder_cost, const int aHold_cost, const
float aLead, const int aSafety, const int aninOrder) :
          pDescription(new String(aDescription)),
          pUnit(new String(aUnit)),
          pCost(aCost),
          pStock(aStock),
          pDemand(aDemand),
          pOrder_cost(aOrder_cost),
          pHold_cost(aHold_cost),
```

```
        pLead(aLead),
        pSafety(aSafety),
        pinOrder(aninOrder)
    {
    if (firstProd == (Gen_product *)NULL)
        {
        firstProd = this;
        lastProd = this;
        pNumber = 1;
        }
    else
        {
        pNumber = lastProd->pNumber + 1;
        lastProd->nextProd = this;
        prevProd = lastProd;
        lastProd = this;
        }
    }


Gen_product::Gen_product(const Gen_product &aProduct) :
        pDescription(new String(*aProduct.pDescription)),
        pUnit(new String(*aProduct.pUnit))

    // Build from existing Product
```

```
    {
    pCost = aProduct.pCost;
    pStock = aProduct.pStock;
    pDemand = aProduct.pDemand;
    pOrder_cost = aProduct.pOrder_cost;
    pHold_cost = aProduct.pHold_cost;
    pLead = aProduct.pLead;
    pSafety = aProduct.pSafety;
    pinOrder = aProduct.pinOrder;

    pNumber = lastProd->pNumber + 1;
    lastProd->nextProd = this;
    prevProd = Gen_product::lastProd;
    lastProd = this;
    }


Gen_product::~Gen_product(void)
    {
    if (firstProd == this)
        {
        firstProd = nextProd;
        nextProd->prevProd = (Gen_product *)NULL;
        }
    else
        if (lastProd == this)
```

```
                {
                lastProd = prevProd;
                prevProd->nextProd = (Gen_product *)NULL;
                }
        else
                {
                nextProd->prevProd = prevProd;
                prevProd->nextProd = nextProd;
                }
    delete pDescription;
    delete pUnit;
    }

Gen_product &Gen_product::operator =(const Gen_product
&aProduct)          // Assign Product
    {
    *pDescription = *aProduct.pDescription;
    *pUnit = *aProduct.pUnit;
    pCost = aProduct.pCost;
    pStock = aProduct.pStock;
    pDemand = aProduct.pDemand;
    pOrder_cost = aProduct.pOrder_cost;
    pHold_cost = aProduct.pHold_cost;
    pLead = aProduct.pLead;
    pSafety = aProduct.pSafety;
    pinOrder = aProduct.pinOrder;
    return(*this);}
```

```
String &Gen_product::description(void) const //  Returns a
                    // pointer to the product description
     {
     return(*pDescription);
     }

String &Gen_product::unit(void) const
     {
     return(*pUnit);
     }

String &Gen_product::description(const char *aDescription)
     {
     return(*pDescription = String(aDescription));
     }

String &Gen_product::unit(const char *aUnit)
     {
     return(*pUnit = String(aUnit));
     }

int Gen_product::cost(void) const
     {
     return (pCost);}
     }
```

```
int Gen_product::stock() const
     {
     return (pStock);
     }

int Gen_product::demand(void) const
     {
     return (pDemand);
     }

int Gen_product::order_cost(void) const
     {
     return (pOrder_cost);
     }

int Gen_product::number(void) const
     {
     return (pNumber);
     }

int Gen_product::safety(void) const
     {
     return (pSafety);
     }
```

```
int Gen_product::inOrder(void) const
     {
     return (pinOrder);
     }

int Gen_product::hold_cost(void) const
     {
     return (pHold_cost);
     }

float Gen_product::lead(void) const
     {
     return (pLead);
     }

int Gen_product::cost(int price)
     {
     pCost = price;
     return (pCost);
     }

int Gen_product::stock(int quantity)
     {
     pStock = quantity;
     return (pStock);
     }
```

```
int Gen_product::demand(int quantity)
     {
     pDemand = quantity;
     return (pDemand);
     }

int Gen_product::order_cost(int amount)
     {
     pOrder_cost = amount;
     return (pOrder_cost);
     }

int Gen_product::safety(int quantity)
     {
     pSafety = quantity;
     return (pSafety);
     }

int Gen_product::inOrder(int quantity)
     {
     pinOrder = quantity;
     return (pinOrder);
     }
```

```
int Gen_product::hold_cost(int amount)
     {
     pHold_cost = amount;
     return (pHold_cost);
     }

float Gen_product::lead(float days)
     {
     pLead = days;
     return (pLead);
     }

ostream &Gen_product::display() const
     {
     return(cout << "\n\n\tProduct Number:\t" << pNumber
               << "\n\tProduct Description\t\t" <<
*pDescription
               << "\n\n\tCost:\t\t\t" << pCost
               << "\tDemand per period\t" << pDemand
               << "\n\tOrdering cost\t\t" << pOrder_cost
               << "\tCurrent stock\t\t"<< pStock
               << "\n\tHolding cost\t\t" << pHold_cost
               << "\tSafety stock\t\t" << pSafety
               << "\n\tEconomic Order Quantity " << eoq()
               << "\tQuantity in Order\t" << pinOrder
```

```
            << "\n\tAnnual inventory costs\t" << aho()
            << "\tLeading time\t\t" << pLead << "\n" );
    }

int Gen_product::sale(int quantity)
    {
    if (quantity > pStock)
        {
    cout <<  "\n\tMay not sell merchandise not in stock\n";
        return(pStock);
        }
    pStock -= quantity;
    order();
    return(pStock);
    }


int Gen_product::supply(int quantity)
    {
    pStock += quantity;
    pinOrder -= quantity;
    if (pinOrder <= 0)
        pinOrder = 0;
    return (pStock);
    }
```

```
int Gen_product::eoq() const
     {
     return (sqrt(2 * pDemand * pOrder_cost / pHold_cost));
     }

int Gen_product::order()
     {
     if (pStock + pinOrder - pSafety <= pLead * pDemand)
          {
          cout << "\n\tPlacing and order of " << eoq() <<
"\n";
          pinOrder += eoq();
          }
     return (pinOrder);
     }

int Gen_product::aho() const
     {
     return (2 * pOrder_cost * pDemand * pHold_cost);
     }

ostream &operator << (ostream &anOstream,const String
&aString)
     {
     return(anOstream << aString.characterString);
     }
```

```
/*
     Function: ostream &operator
          << (ostream &,const Gen_product &aProduct)
     Purpose:  Send the given Product instance to the given
     output stream
*/
ostream &operator << (ostream &anOstream,const Gen_product
&aProduct)
     {
     return(anOstream << aProduct.pNumber << ' ' <<
          *aProduct.pDescription);
     }


Gen_product::listProducts(void)
     {
     int count = 0;
     Gen_product *aProduct;
     if (firstProd == (Gen_product *)NULL)
          return(count);
     aProduct = firstProd;
     while (aProduct != (Gen_product *)NULL)
          {
          cout << aProduct->display();
          aProduct = aProduct->nextProd;
          }
     return(count);
     }
```

```
/*
     Program:          STRINGS.CPP
     Author:           Dr. Brian Le Claire
                       This code was adapted by Antonio MP Reis
                       from the "C_EX_09.CPP" file provided to the
                       students of the 216-746 course at UWM
                       (Spring 1992)
     Purpose:  Supporting the String class
*/
#include "c:\trab\thesis\gen_prod.h"
static char dummyCharacter = '\0';
size_t String::constructFrom(const char *aString)
     {
     size_t characterCount = strlen(aString);

     if(characterString != aString)
          {
          if(characterString != (char *)NULL)
               {
               delete characterString;
               }
          characterString = new char[characterCount + 1];
          strcpy(characterString,aString);
     }
     return(characterCount);                    // Answer size
     }
```

```
Relational String::compare(const String &aString) const
     {
     int result = strcmp(characterString,
                    aString.characterString);
     if(result)                    // Non-zero for < or >
          {
          if(result < 0)           // Less than if < 0
               return(lessThan);
          return(greaterThan);          // Greater than if>0
          }
     return(equalTo);              // Equal to if 0
     }


String::String() : characterString((char *)NULL)
     {
     constructFrom("");  // Build from empty character array
     }

/*
     Function: String(const char *)

     Purpose:  This constructor builds a String based on a C
               character array
*/
```

```
String::String(const char *aString) : characterString((char
*)NULL)
     {
     constructFrom(aString);  // Build from character array
     }
String::String(const String &aString) :
characterString((char *)NULL)      {
     constructFrom(aString.characterString);
     }

String::~String()
     {
     delete characterString;
     }

String &String::operator =(const String &aString)
     {
     constructFrom(aString.characterString);
     return(*this);
     }

String::operator char *() const
     {
     return(characterString);
     }
```

```
size_t String::length() const
     {
     return(strlen(characterString));
     }
String &String::concatenate(const String &aString)
     {
     size_t characterCount = length();
     char *combineCharacters = new char[characterCount +
          aString.length() + 1];
     strcpy(combineCharacters,characterString);
     strcpy(combineCharacters +
          characterCount,aString.characterString);
     delete characterString;
     characterString = combineCharacters;
     return(*this);
     }
String &String::copy(const String &aString)
     {
     constructFrom(aString.characterString);
     return(*this);
     }
Boolean String::isEqualTo(const String &aString) const
     {
     if(compare(aString) == equalTo)
          return(True);
     return(False);
     }
```

```
Boolean String::isLessThan(const String &aString) const
    {
    if(compare(aString) == lessThan)
        return(True);
    return(False);
    }

Boolean String::isGreaterThan(const String &aString) const
    {
    if(compare(aString) == greaterThan)
        return(True);
    return(False);
    }

char &String::at(const size_t index) const
    {
    if(length() <= index)
        return(dummyCharacter);
    return(*(characterString + index));
    }
```

```
/*
     Program:  BAC_PROD.CPP
     Author:        Antonio MP Reis
     Date:          April, 20, 1993
     Purpose:  Supporting the backlogged products behavior

*/

#include "c:\trab\thesis\bac_prod.h"

Back_product::Back_product(const char *aDescription, const
char *aUnit, const int aCost, const int aStock, const int
aDemand, const int aOrder_cost, const int aHold_cost, const
float aLead, const int aSavety, const int aninOrder, const
int aBack_cost) : Gen_product(aDescription, aUnit, aCost,
aStock, aDemand, aOrder_cost, aHold_cost, aLead, aSavety,
aninOrder), pBack_cost(aBack_cost)
     {    }
Back_product::Back_product(const Back_product &aProduct) :
          Gen_product((Gen_product &)aProduct)

          // Build from existing Product

     {
     pBack_cost = aProduct.pBack_cost;
     }
```

```
Back_product::~Back_product(void)
    {
    }

Back_product &Back_product::operator =(const Back_product
&aProduct)          // Assign Product
    {
    description(aProduct.description());
    unit(aProduct.unit());
    cost(aProduct.cost());
    stock(aProduct.stock());
    demand(aProduct.demand());
    order_cost(aProduct.order_cost());
    hold_cost(aProduct.hold_cost());
    lead(aProduct.lead());
    safety(aProduct.safety());
    inOrder(aProduct.inOrder());
    back_cost(aProduct.back_cost());
    return(*this);
    }

int Back_product::back_cost(void) const
    {
    return (pBack_cost);
    }
```

```
int Back_product::back_cost(int amount)
     {
     pBack_cost = amount;
     return (pBack_cost);
     }

ostream &Back_product::display() const
     {
     return(cout << "\nBACKLOGGED PRODUCT\n\n\tProduct
Number:\t"     << number()
               << "\n\tProduct Description\t\t" <<
description()
               << "\n\n\tCost:\t\t\t" << cost()
               << "\tDemand per period\t" << demand()
               << "\n\tOrdering cost\t\t" << order_cost()
               << "\tCurrent stock\t\t"<< stock()
               << "\n\tHolding cost\t\t" << hold_cost()
               << "\tQuantity in Order\t" << inOrder()
               << "\n\tBacklogging cost\t" << back_cost()
               << "\tLeading time\t\t" << lead()
          << "\n\tEconomic Order Quantity " << eoq()
          << "\n\tQuantity to backlog\t" << dif()
          << "\n\tAnnual inventory costs\t" << aho()
          << "\n" );
     }
```

```
int Back_product::sale(int quantity)
     {
     stock(stock() - quantity);
     order();
     return(stock());
     }

int Back_product::eoq() const
     {
     return (sqrt(2 * demand() * order_cost() / hold_cost()
*    (pBack_cost + hold_cost()) / pBack_cost));
     }

int Back_product::order()
     {
     if (stock() + inOrder() + dif() <= lead() * demand())
          {
          cout << "\n\tPlacing and order of " << eoq() <<
"\n";
          inOrder(inOrder() + eoq());
          }
     return (inOrder());
     }
```

```
int Back_product::aho() const
     {
     return (demand() / eoq() *
          ( order_cost() + hold_cost() / 2 * ( dif())^2 /
demand() + pBack_cost / 2 * maxStk()^2 / demand()));
     }

int Back_product::dif(void) const
     {
     return (eoq() - maxStk());
     }

int Back_product::maxStk(void) const
     {
     return (sqrt(2 * demand() * order_cost() / hold_cost()
/    (pBack_cost + hold_cost()) * pBack_cost));
     }
```

```
/*
     Program: GEN_PROD.H
     Author:  Antonio M.P. Reis (String class based in Dr
Leclaire string class)
     Purpose: Header file for products
*/
#ifndef __GEN_PROD_H
/*
      To avoid multiple inclusion
*/
#define __GEN_PROD_H
#include <iostream.h>    // Required for use of cout
extern "C"
{
#include <string.h> // Required for string manipulation
functions #include <stdlib.h>      // abs
#include <math.h>       // sqrt
}

/*
     Boolean enumeration
*/
```

```
enum Boolean
     {
     False,
     True
     };

enum Relational
{
          equalTo  =  0,
        lessThan   = -1,
     greaterThan   =  1
};

/*
     Class:          String

     Purpose:  This class provides the user with general
               purpose string handling capabilities.
*/
```

```
class String
{
     friend ostream &operator <<(ostream &,const String &);
protected:
          char *characterString;    // Character string
          size_t constructFrom(const char *);
Relational compare(const String &) const;
          String(void);          // Build empty string
          String(const char *);
          String(const String &); // Build from existing
          ~String(void);         // Release dynamic memory
          String &operator =(const String &);
          operator char *(void) const;
          size_t length(void) const;
          String &concatenate(const String &),
            &copy(const String &);        // Copy
          Boolean isEqualTo(const String &) const, // Equal?
               isLessThan(const String &) const, // Less?
               isGreaterThan(const String &) const;
char &at(const size_t) const; // Allow changes and queries
};


ostream &operator <<(ostream &,const String &);
```

```
// ---------------- A Generic Product class
class Gen_product
     {
     friend ostream &operator <<(ostream &,const Gen_product
&);
     private:
          static Gen_product  *firstProd;
          static Gen_product  *lastProd;
          String              *pDescription;
          String              *pUnit;
          int            pNumber, pCost, pStock, pDemand;
          int            pOrder_cost, pHold_cost, pSafety;
          int                      pinOrder;
          float                    pLead;
          Gen_product         *nextProd;
          Gen_product         *prevProd;
     public:
          Gen_product(const char * = "", const char * = "",
               const int = 0, const int = 0, const int = 0,
               const int = 0, const int = 0, const float =0,
               const int = 0, const int = 0);
          Gen_product(const Gen_product &);
          ~Gen_product(void);
          Gen_product &operator =(const Gen_product &);
          String &description(void) const,
                 &unit(void) const;
          int cost(void) const;
```

```
        int stock(void) const;
        int demand(void) const;
        int order_cost(void) const;
        int hold_cost(void) const;
        int safety(void) const;
        int inOrder(void) const;
        int number(void) const;
        float lead(void) const;
        String &description(const char *),
               &unit(const char *);
        int cost(int amount),
            stock(int quantity),
            demand(int quantity),
            order_cost(int amount),
            hold_cost(int amount),
            safety(int quantity),
            inOrder(int quantity);
        float lead(float days);
        virtual int sale(int quantity);
        virtual int supply(int quantity);
        virtual ostream &display() const;
        static int listProducts(void) const;
        virtual int eoq(void) const;
        virtual int order(void);
        virtual int aho(void) const;};
ostream &operator <<(ostream &,const Gen_product &);
#endif
```

```
/*
     Program: BAC_PROD.H
     Author:  Antonio M.P. Reis
     Purpose: Header file for backlogged products
*/

#ifndef __BAC_PROD_H

/*
      To avoid multiple inclusion
*/

#define __BAC_PROD_H

# include <iostream.h>
# include "c:\trab\thesis\gen_prod.h"

// ----------------- A  Backlogged Product class


class Back_product : public Gen_product
     {
     private:
          int            pBack_cost;
     public:
          Back_product(const char * = "", const char * = "",
```

```
            const int = 0, const int = 0, const int = 0,
            const int = 0, const int = 0, const float =0,
            const int = 0, const int = 0, const int = 0);
        Back_product(const Back_product &);
        ~Back_product(void);
        Back_product &operator =(const Back_product &);
        int back_cost(void) const;
        int back_cost(int amount);

        virtual ostream &display() const;
        virtual int sale(int quantity);
        virtual int eoq(void) const;
        virtual int order(void);
        virtual int aho(void) const;
        virtual int dif(void) const;
        virtual int maxStk(void) const;
    };
#endif
```

BIBLIOGRAPHY

Gould, F.J., Eppen, G.D., Schmidt, C.P., "Introductory Management Science", Prentice Hall, 1991, pp 479-542.

Hanssmann, Fred, "Operations Research in Production and Inventory Control", John Wiley & Sons, Inc., 1962. (HD 55 .H2)

Jain, Hemant, "Reading packet for the 216-749 Database Management course", University of Wisconsin Milwaukee, Spring 1993.

Johnsonbaugh, R., Kalin, M. "Applications programming in ANSI C", Macmillan, 1990.

Korson, T. and McGregor, J.D., "Understainding Object-Oriented: A Unifying Paradigm," CACM, 33 (9), Sept 1990, 40-60.

Leclaire, Brian, "Reading Packet for Spring 1992 - 216-746", University of Wisconsin Milwaukee Student Association, 1992.

Naddor, Elizer, "Inventory Systems", John Wiley & Sons, Inc., 1966. (HD 55 .N3 c.2)

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen W., "Object-Oriented Modeling and Design", Prentice Hall, 1991.

Stevens, Al, "Teach yourself C++", MIS Press, New York, 1991.